



CHAPTER 3

New Challenges and New Threats

Now that we know what Web Services are (Chapter 1) and have at least a basic understanding of the principles of security (Chapter 2), we are in a position to answer the question “what kind of security does Web Services need?” We will see in this chapter that Web Services security focuses on the application layer, although security at the lower layers remains important. The principles of security are the same as those we encountered in the previous chapter: authentication, authorization, and so forth. The implementation technologies on which we focus are HTTP and SOAP, although we will keep SMTP security in mind also since SOAP can be bound to SMTP as well as HTTP.

It may not seem immediately obvious why security for SOAP presents such a challenge. After all, SOAP is generally bound to HTTP, which already has SSL for authentication and confidentiality. In addition, many Web authorization tools already exist. It is a reasonable question to ask why these aren’t enough, and the answer is made up of a number of reasons.

The first reason is that, although frequently bound to HTTP, SOAP is independent of the underlying communications layers. Many different communications technologies can be used in the context of one multihop SOAP message; for example, using HTTP for the first leg, then SMTP for the next leg, and so forth. End-to-end security cannot therefore rely on a security technology that presupposes one particular communications technology. Even in the case of a single SOAP message targeted at a Web Service, transport-level security only deals with the originator of the SOAP request. SOAP requests are generated by machines, not by people. If the Web Service wishes to perform security based on the end user, it must have access to authentication and/or authorization information about the end user on whose behalf the SOAP request is being sent. This is the second reason for Web Services security.

This information is not available in the transport layer, which deals only with the originator of the SOAP request. When SOAP messages are routed between Web Services, the same problem applies. The security context spans multiple connections, meaning the principles of security such as integrity and confidentiality must also apply across these multiple connections. These challenges are met by persisting security information inside the SOAP message. This chapter introduces WS-Security, a framework for including security information as XML in SOAP messages. Next, the specifications for expressing security information (digital signatures, encryption, authentication, and authorization data) in XML are introduced.

If confidentiality, integrity, and identity-based security can be viewed as the positive aspects of security, then protecting against hacker attacks is the negative aspect. Hacker attacks are a fact of life when computers connect to the Internet. These attacks tend to follow the path of least resistance; that is, by circumventing security, not tackling it head-on. A sophisticated authentication system is useless if it requires people to “play by the rules” and these rules can be bypassed. Now that many of the vulnerabilities at lower layers of the network have been addressed, the playing field has moved to the application layer. In this chapter, we’ll see how these attacks share many characteristics with the older, more traditional attacks at lower layers of the communications stack.

Web Services presents both a security challenge and a security threat. The challenge is to implement the principles of security at the application layer. The threat is that

Web Services presents a new avenue of attack into enterprise systems, one that is not addressed by current security infrastructure (including firewalls). This chapter examines the new technologies that address these challenges and threats.

WEB SERVICES SECURITY CHALLENGES

In Chapter 2, we were introduced to the layers of the OSI stack. Table 3-1 shows us how the OSI layers apply to Web Services.

Notice that SOAP is in layer 7, together with HTTP and SMTP. However, SOAP travels over HTTP or SMTP. This does not mean that SOAP belongs in a new layer, a layer 8. On the contrary, the seven-layer communications stack still applies for *each individual communication* from a SOAP requester to a Web Service. However, one SOAP-based communication is not the full story. Web Services security presents three challenges:

- The challenge of security based on the end user of a Web Service
- The challenge of maintaining security while routing between multiple Web Services
- The challenge of abstracting security from the underlying network

Let's examine each of these challenges.

The Challenge of Security Based on the End User of a Web Service

SOAP is a technology used to enable software to talk to other software much easier than was previously possible. End users (that is, humans) do not make SOAP messages themselves. However, if access to the Web Service is to be decided based on the information about the end user, the Web Service must have access to the information that allows it to make this authorization decision. This information does not have to include the end user's actual identity. Consider Figure 3-1.

Layer Number	Layer Name	Web Services Technology
Layer 7	Application	HTTP, SMTP, SOAP
Layer 6	Presentation	Encrypted data, Compressed data
Layer 5	Session	POP/25, SSL
Layer 4	Transport	TCP, UDP
Layer 3	Network	IP Packets
Layer 2	Data Link	PPP, 802.11, etc.
Layer 1	Physical	ADSL, ATM, etc.

Table 3-1. OSI Layers

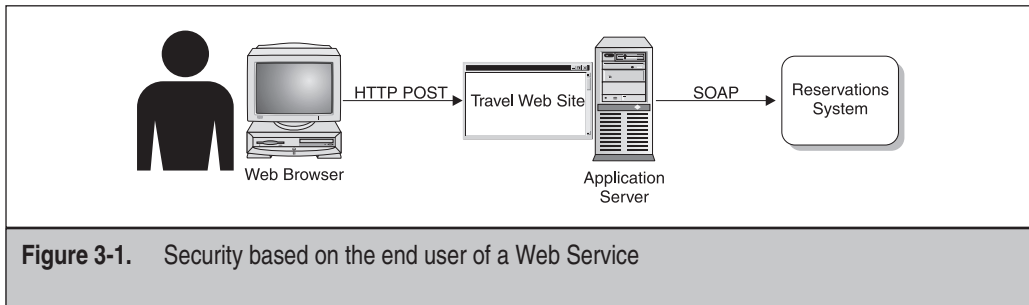


Figure 3-1. Security based on the end user of a Web Service

The end user in Figure 3-1 is accessing a travel Web site and making a reservation. The reservation is made, on the user's behalf, on a third-party system accessed using SOAP. The end user may have authenticated to the travel Web site, perhaps using a username and password. Because of the successful authentication, the user may have been shown personalized content. Information about the user's identity, as well as attributes of the end user such as their travel preferences and previous bookings, are known to the Web site. However, in Figure 3-1, we see that the Web Service only has visibility of the travel Web site, not the end user.

How can this information about the end user be conveyed to the Web Service? Session layer or transport layer security between the application server and the Web Service doesn't convey information about the identity of the end user of the Web Service. It merely conveys information about the application server that is sending the SOAP message. It may be the case that many of the requests to the Web Service originate from that application server.

This challenge is addressed by including security information about the end user in the SOAP message itself. This information may concern the end user's identity, attributes of the end user, or simply an indication that this user has already been authenticated and/or authorized by the Web server. This information allows the Web Service to make an informed authorization decision.

This scenario is likely to be widespread where many Web Services are used to implement functionality "behind the scenes." It shouldn't be the case that the end user has to reauthenticate each time a SOAP request must be sent on their behalf. The challenge of providing this functionality is sometimes called "single sign-on" or "federated trust."

End-User Access to a Web Service: A Practical Example

This example uses the ASP.NET Web Matrix tool, freely downloadable from <http://www.asp.net>. ASP.NET Web Matrix in turn requires the .NET Framework to be installed. The .NET Framework can be downloaded from the following URL: <http://msdn.microsoft.com/netframework>. Please ensure that you download the latest service pack for the .NET Framework.

Simple "Add" and "Subtract" Web Services

When ASP.NET Matrix is opened, it displays the dialog box seen in Figure 3-2.

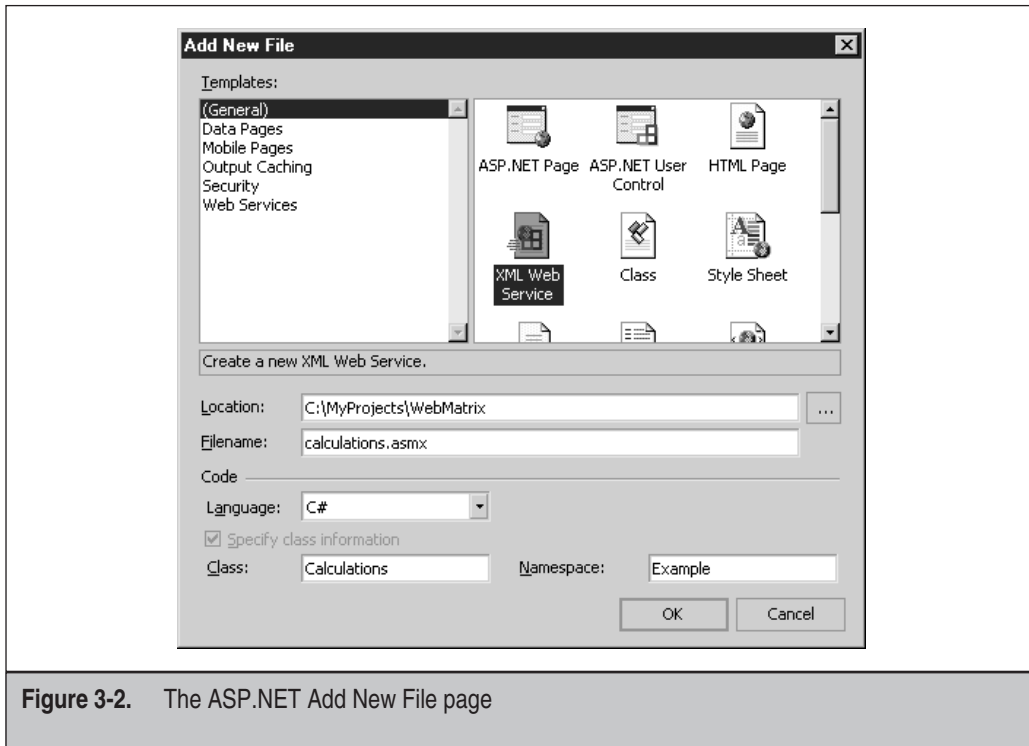


Figure 3-2. The ASP.NET Add New File page

Enter **calculations.asmx** in the Filename text box, **Calculations** in the Class text box, and **Example** in the Namespace text box. Choose C# as the language. Now click OK. You will be presented with the screen shown in Figure 3-3.

Conveniently, for our example, the code for an example Add Web Service is already provided. Now add a “Subtract” method so that the code listing onscreen is identical to the following code:

```
<%@ WebService language="C#" class="Calculations" %>
using System;
using System.Web.Services;
using System.Xml.Serialization;
public class Calculations {
    [WebMethod]
    public int Add(int a, int b) {
        return a + b;
    }

    [WebMethod]
    public int Subtract(int a, int b) {
        return a - b;
    }
}
```

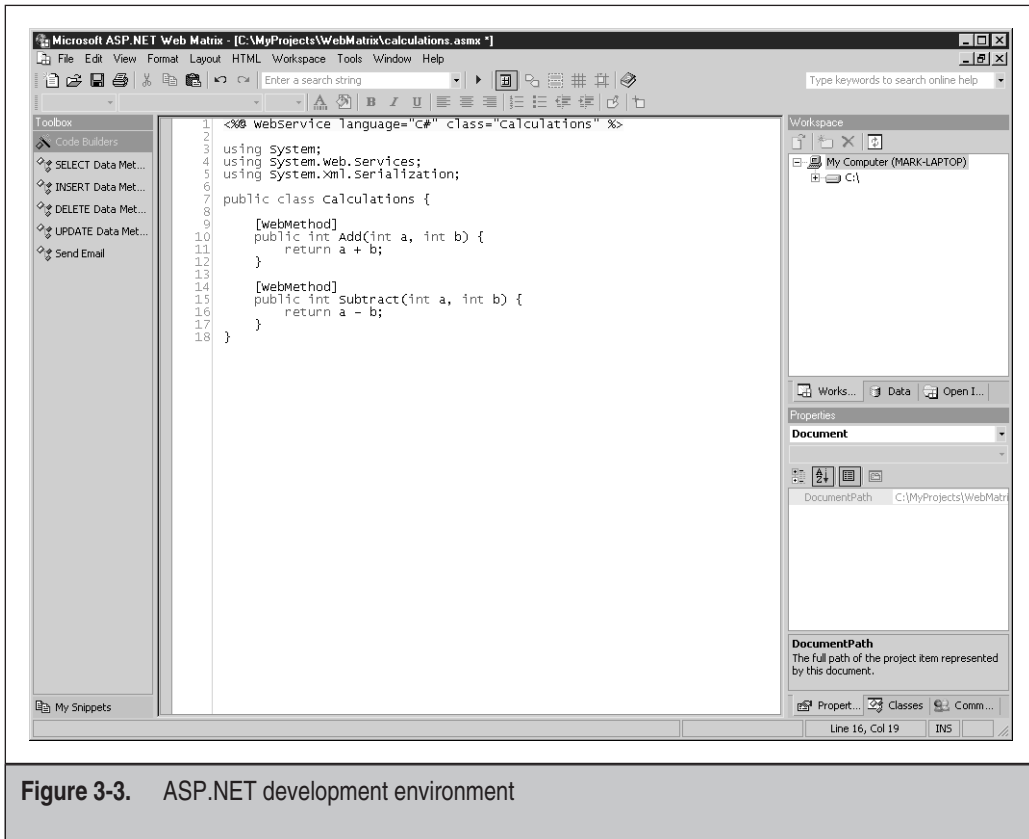


Figure 3-3. ASP.NET development environment

Click the Save button and then click the Run icon (the right-facing triangle on the toolbar). You will be prompted with the dialog box seen in Figure 3-4.

Click the Start button to start the ASP.NET local Web server. Now open a Web browser and navigate to <http://localhost:8080/calculations.aspx>. A Web form is automatically created in order to pass data to the calculation Web Services. In order to see the WDSL descriptors for the Web Service, navigate to <http://localhost:8080/calculations.aspx?WDSL>.

From the main Calculations Web Service page, click the Add link. The resulting page shows a Web form that can be used to submit data to the Web Service, and examples of SOAP messages that can be targeted to the Web Service.

It is important to note the distinction between the user calling the Web Service directly, using the form, and the scenario where SOAP is sent on the user's behalf. This distinction is shown in Figure 3-5.

When the data from the form is submitted to the Web Service, the user's browser is making a direct connection to the Web Service, using HTTP GET or HTTP POST. SSL, or cookies can be used to secure the connection between the Web Service and the user's

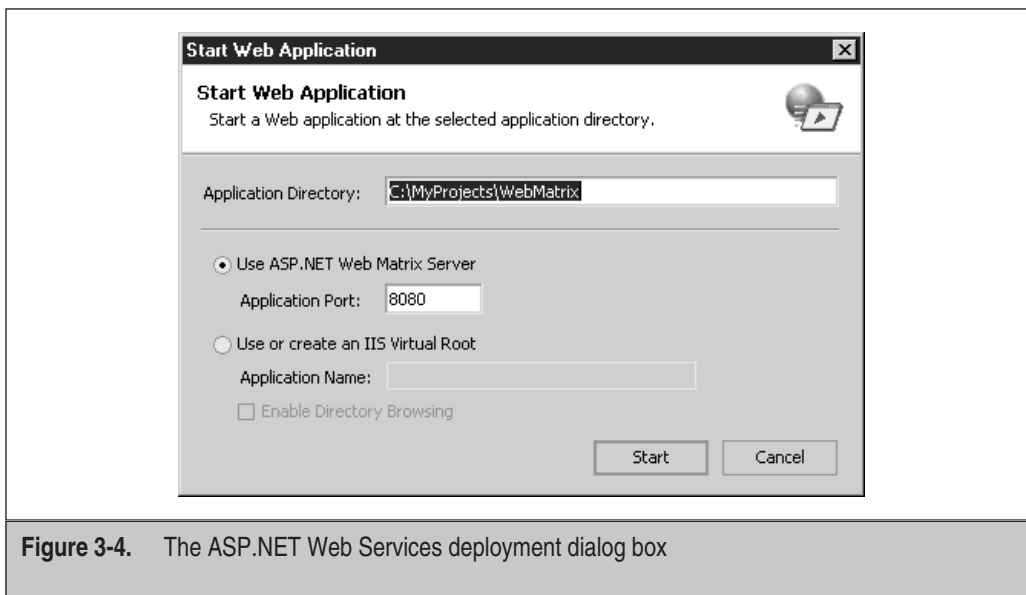


Figure 3-4. The ASP.NET Web Services deployment dialog box

browser. However, when SOAP is used—meaning that one of the example SOAP messages is submitted to the Web Service—the user is one step away from the communication. The application sending the SOAP message on the user’s behalf may use cookies or SSL, but that would only secure the connection between the application and the Web Service. If the user has connected to a Web site that uses the calculation Web Service on their behalf, then the Web Service will only have visibility of the Web

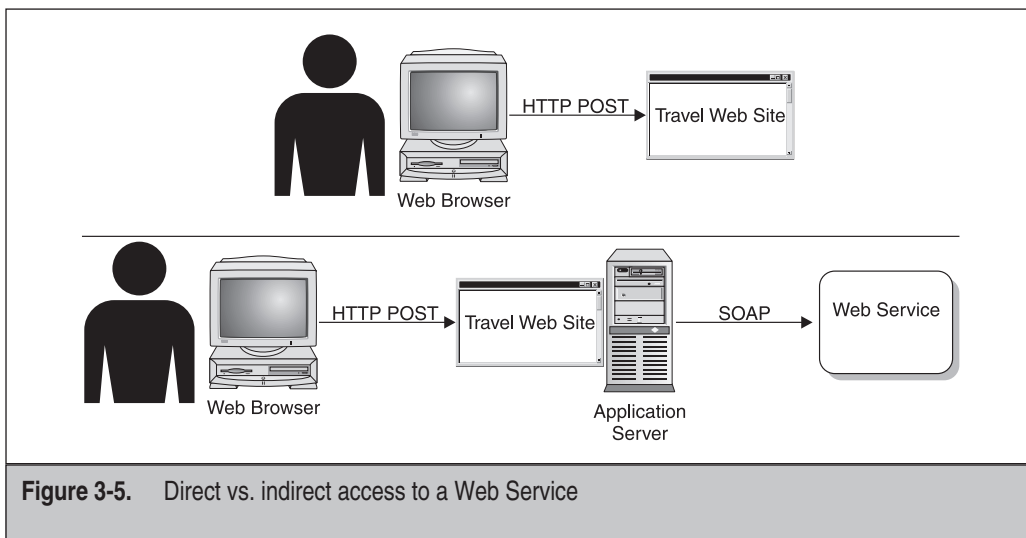


Figure 3-5. Direct vs. indirect access to a Web Service

site that is sending the SOAP message, not of the end user on whose behalf the message is being sent.

The problem is that there are two security contexts in play. These are spelled out in Figure 3-6.

In our trivial calculation example, it may not seem important who is running the Web Service. But imagine that the end user is a currency dealer who connects to a local portal in order to execute a trade. If a Web Service is run on that dealer's behalf, the provider of the Web Service must know not only what portal is sending the SOAP request to it, but *who the dealer is*. The solution to this problem, as we will see, is to include information about the end user in the SOAP message itself.

The Challenge of Maintaining Security While Routing Between Multiple Web Services

Although SOAP routing is not in the scope of SOAP 1.1 or SOAP 1.2, it has been proposed as part of Microsoft's GXA (Global XML Web Services Architecture). WS-Routing provides a means for SOAP messages to route between multiple Web Services. WS-Routing defines how to insert routing information into the header of a SOAP message. This routing information can be thought of as equivalent to routing tables that operate at lower layers of the OSI stack for routing IP packets.

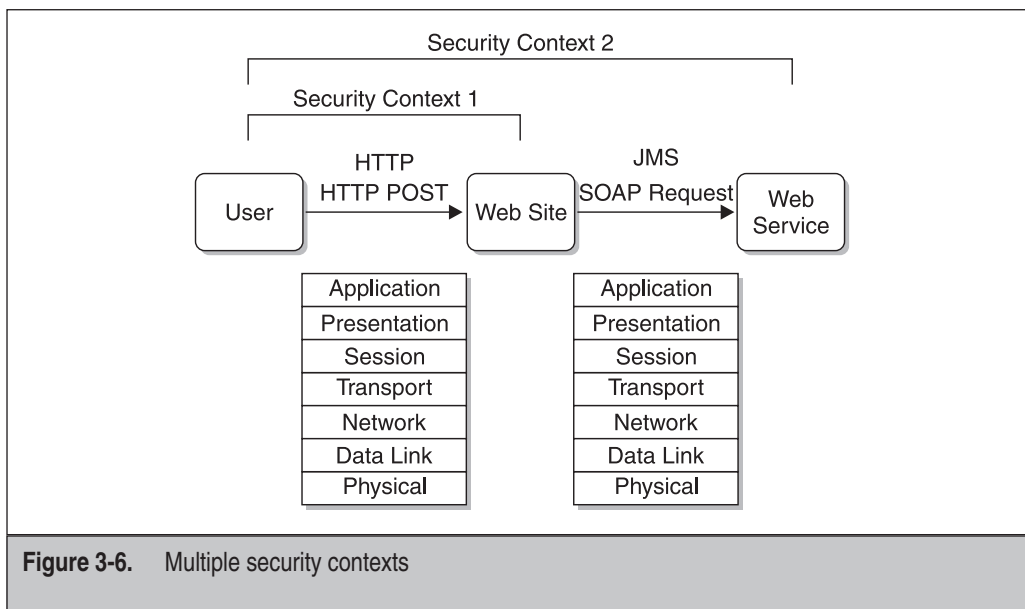


Figure 3-6. Multiple security contexts

WS-Routing means that one SOAP message may traverse multiple SOAP “hops” between the originator and the endpoint. The systems that implement these hops may have nothing in common apart from the ability to parse and route a SOAP message.

The following code listing is an example of a SOAP message that uses WS-Routing in order to route between Web Services. It routes from the originator, via an intermediary, to an endpoint. It is targeted at the Calc Web Service, familiar from the previous example in this chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <h:path xmlns:h="http://schemas.xmlsoap.org/rp/"
      SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next"
      SOAP-ENV:mustUnderstand="1">
      <rp:action xmlns:rp="http://schemas.xmlsoap.org/rp/"
        Addition
      </rp:action>
      <rp:to xmlns:rp="http://schemas.xmlsoap.org/rp/"
        http://www.example.com/Calc
      </rp:to>
      <rp: fwd xmlns:rp="http://schemas.xmlsoap.org/rp/"
        <rp:via>http://www.intermediary.com/webservice</rp:via>
      </rp: fwd>
      <rp:rev xmlns:rp="http://schemas.xmlsoap.org/rp/"
        <rp:via/>
      </rp:rev>
      <rp:from xmlns:rp="http://schemas.xmlsoap.org/rp/"
        originator@example.com
      </rp:from>
      <rp:id xmlns:rp="http://schemas.xmlsoap.org/rp/"
        uuid:EC823E93-BE2B-F9DC-8BB7-CD54B16C6EC1
      </rp:id>
    </h:path>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <SOAPSDK1:Add xmlns:SOAPSDK1="http://tempuri.org/message/">
      <A>1</A><B>2</B>
    </SOAPSDK1:Add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3-7 illustrates how this routing scenario involves more than one security context.

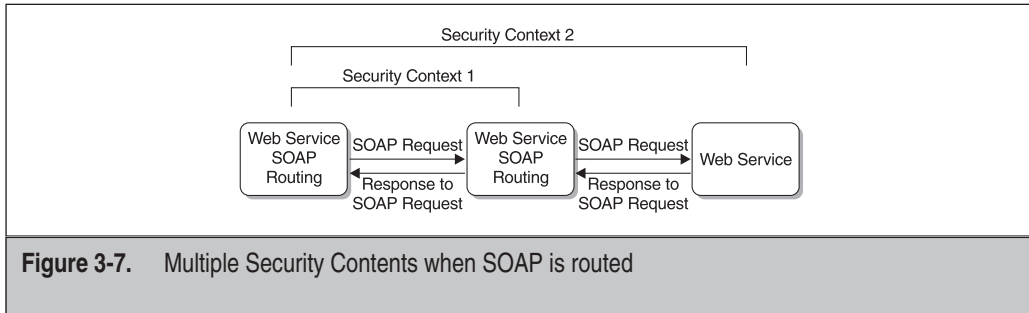


Figure 3-7. Multiple Security Contexts when SOAP is routed

SOAP routing does not have to depend on routing information in the SOAP message itself. Routing can be performed for a number of reasons, including scaling a Web Services infrastructure between multiple SOAP servers, bridging between two different networking protocols, or transforming message content from one format to another. All of these scenarios extend the security context beyond a single SOAP request/response.

When routing between Web Services, the requirement for confidentiality can apply from the originator through to the final SOAP endpoint. It may be a requirement that information be kept secret from SOAP intermediaries. There may be a chance that intermediaries may disclose the information either deliberately or through leaving “gaps” between one transport-level security session and the next. While the data is decrypted, it is vulnerable. This is the same problem that plagued the first release of the Wireless Access Protocol (WAP), in which data was decrypted in between the wireless encryption session and encryption on the fixed wire. This so-called “WAP gap” caused a loss of confidence in WAP security and was addressed in later releases of the WAP specification. Implementing encryption only at the transport level makes a “SOAP gap.”

It is often noted that most security breaches happen not while data is in transit, but while data is in storage. This is the principle of least resistance—attempting to decrypt eavesdropped encrypted data from an SSL session is much more difficult than simply testing if a Web site maintainer has remembered to block direct access to the database where the credit card numbers are stored. If decrypted data is stolen from a database, the consequences are no less dramatic. Once data has reached its final destination, it must be stored in a secure state. Confidentiality for a SOAP transaction should not involve simply chaining instances of confidentiality together, since “SOAP gaps” of unencrypted data are available between each decryption and encryption.

The Challenge of Abstracting Security from the Underlying Network

As we saw in Chapter 1, the term “Web” Services is misleading. “Services” is somewhat redeemable since it indicates a services-oriented architecture (SOA).

However “Web” points squarely at the World Wide Web. However, Web Services is not reliant on HTTP. “Net Services” would have been a better term, but it is too late to change the name now. Just as Web Services is not reliant on the Web, Web Services security cannot rely on Web security. This does not mean that Web security can be ignored. We’ll see later in this chapter that Web server security can be the “soft underbelly” of an HTTP-based SOAP service and may well afford the path of least resistance to an attacker.

SSL is the obvious choice for confidentiality and authentication (one-way or two-way) for the single connection between a SOAP requester and a Web Service over HTTP. First-generation Web Services are almost unanimously using HTTP, so for the implementer, the reasons to use SSL are compelling. However, second-generation Web Services are likely to move beyond HTTP to reliable messaging frameworks such as HTTPR and SonicXQ, and peer-to-peer technologies such as Jabber.

SSL: A Pragmatic Solution

The challenge of HTTP independence is one that faces standards groups as they are generating Web Services security specifications. A more mundane challenge faces the architect charged with implementing Web Services for their organization: to get secure services up and running as soon as possible. SSL provides for confidentiality between the SOAP requester and the Web Service itself, as well as authentication. If this is all that is required, with no possibility that end-user security or SOAP routing will be introduced to the solution in the future, then this is a pragmatic solution. Chapter 2 explained that it is the high-level principles of security that must be implemented, and just because SOAP is used, it doesn’t follow that SOAP security must be used. SSL is available in all Web servers, and with the vast majority of first-generation Web Services using HTTP, it is a useful and pragmatic solution.

MEETING THE CHALLENGES: NEW TECHNOLOGIES FOR WEB SERVICES SECURITY

At this point, we’ve seen the requirement for new Web Services security specifications. Let’s look at how this requirement is being met by introducing new security specifications, which will be explored in greater detail in subsequent chapters.

Persistent Security

The three security challenges we’ve seen have one thing in common. The principles of security must apply to a security context that includes more than a single request/response SOAP message. The solution to this problem is to *persist* security data in the SOAP message itself. The security data therefore is not lost after one SOAP communication has ended. Confidential information in a SOAP message should remain confidential over the course of a number of SOAP hops.

A number of industry specifications have been developed for this purpose. These specifications can be organized into two distinct categories:

1. A standardized framework to include XML-formatted security data into SOAP messages.
2. Standards for expressing security data in XML format. This security information should be used for the high-level principles of security: confidentiality, authentication, authorization, integrity, and so forth.

Including XML-Formatted Security Data in SOAP Messages: Introducing WS-Security

WS-Security has emerged as the de facto method of inserting security data into SOAP messages. Work on WS-Security began in 2001, was published by Microsoft, VeriSign, and IBM in April 2002, and was then submitted in June 2002 to the OASIS standards body in order to be made into an industry standard. WS-Security defines placeholders in the SOAP header in order to insert security data. It defines how to add encryption and digital signatures to SOAP messages, and then a general mechanism for inserting arbitrary security tokens. WS-Security is “tight” enough to present the definitive means of including security data into SOAP messages, but is “loose” enough to not place limits on what that security data can be.

Confidentiality for Web Services: Introducing XML Encryption

XML Encryption is a specification from the W3C. It provides not only a way of encrypting portions of XML documents, but also a means of encrypting any data and rendering the encrypted data in XML format. XML Encryption makes encryption functionality easier to deploy.

XML Encryption is not a replacement for SSL. SSL is still the de facto choice for confidentiality between two entities that are communicating using HTTP. However, if the security context extends beyond this individual HTTP connection, XML Encryption is ideal for confidentiality. The capability to encrypt XML is nothing new, because XML is just text after all. However, the ability to *selectively encrypt* XML data is what makes XML Encryption so useful for Web Services. Encrypting an entire SOAP message is counterproductive, because the SOAP message must include enough information to be useful—routing information, for example. Selectively encrypting data in the SOAP message is useful, however. Certain information may be hidden from SOAP intermediaries as it travels from the originator to the destination Web Service.

XML Encryption does not introduce any new cryptography algorithms or techniques. Triple-DES or RSA encryption may still be used for the actual encryption. XML Encryption provides a way to format the meta-information about which algorithm was used, and when the encryption occurred. This aids the Web Service in decrypting the data, provided the decryption key is available to it. This is important, because prior to XML

Encryption the only standardization of encryption data was for e-mail messages (that is, S/MIME). If an organization wished to send encrypted data to another organization, both organizations would have to agree on the format of the encrypted data, how and which algorithms to use, and possibly also how to send an encrypted key. Now that information can be contained in an XML Encryption block. Chapter 5 explores XML Encryption in detail, including code examples in C# and Java.

WS-Security defines how XML Signature data can be included in a SOAP message. This provides *persistent confidentiality* beyond a single SOAP communication.

Integrity for Web Services: Introducing XML Signature

XML Signature is a specification produced jointly by the W3C and the Internet Engineering Task Force (IETF). Like XML Encryption, it does not only apply to XML. As well as explaining how to digitally sign portions of an XML document, XML Signature also explains how to express the digital signature of any data as XML. As such, it is an “XML-aware digital signature.” PKCS#7 is a means of rendering encrypted data, and signed data, which predates XML Signature and XML Encryption. Rather than using XML, it uses Abstract Syntax Notation number 1 (ASN.1). ASN.1 is a binary format, renowned for its complexity. Producing or verifying a PKCS#7 signature requires not just cryptography software, but also an ASN.1 interpreter. XML Signature also requires cryptography software, of course, but an XML DOM replaces the ASN.1 interpreter.

The power of XML Signature for Web Services is the ability to selectively sign XML data. For example, a single SOAP parameter passed to a method of a Web Service may be signed. If the SOAP request passes through intermediaries en route to the destination Web Service, XML Signature ensures end-to-end integrity.

WS-Security describes how to include XML Signature data in a SOAP message. An important feature of XML Signature is that it can be very selective about what data in an XML instance is signed. This feature is particularly useful for Web Services. For example, if a single SOAP parameter needs to be signed but the SOAP message’s header needs to be changed during routing, an XML Signature can be used that only signs the parameter in question and excludes other parts of the SOAP message. Doing so ensures end-to-end integrity for the SOAP parameter while permitting changes to the SOAP’s header information. Chapter 4 explores XML Signature in detail, including code examples in C# and Java.

Web Services Authentication and Authorization: Introducing SAML, XACML, Passport, and Liberty

Single sign-on (SSO) is one of the “hard” problems in information technology. As seen in Figure 3-5, if a user signs on to a Web site and then a SOAP request is produced on the user’s behalf, the destination Web Service may require information about the end user in order to make an authorization decision. Otherwise, the destination Web Service only has visibility of the machine that is creating the SOAP request. There are two approaches to this requirement. The first approach is to include the information in

the SOAP message itself. The second approach is to request this information from a central repository.

Security Assertions Markup Language (SAML) provides a means of expressing information about authentication and authorization, as well as attributes of an end user (for example, a credit limit) in XML format. SAML data may be inserted into a SOAP message using the WS-Security framework. SAML is used to express information about an act of authentication or authorization that has occurred in the past. It does not provide authentication, but can express information about an authentication event that has occurred in the past; for example, "User X authenticated using a password at time Y." If an entity is authorized based on the fact that they were previously authorized by another system, this is called "portable trust." SAML is important to address the challenge of multihop SOAP messages also, because separate authentication to each Web Service is often out of the question. By authenticating once, being authorized, and effectively reusing that authorization for subsequent Web Services, single sign-on for Web Services can be achieved.

Note that this information in a SAML assertion may not indicate the end user's identity. The user may have authenticated using a username and password, and the administrator of the Web site may have no idea of the user's actual identity. It may simply be an indication that the user presented credentials and was authenticated and authorized. SAML allows information to be placed into a SOAP message to say "this person was authorized according to a certain security policy at a certain time." If the recipient of this SOAP message trusts the issuer of the SAML data, the end user can also be authorized for the Web Service. This SAML data is known as an "assertion" because the issuer is *asserting* information about the end user. The concept of security assertions has existed before SAML, and is already widely used in existing software.

XML Access Control Markup Language (XACML) is designed to express access control rules in XML format. Although the two technologies are not explicitly linked, XACML may be used in conjunction with SAML. An authorization decision expressed in a SAML assertion may have been based on rules expressed in XACML.

Microsoft's Passport technology takes a different approach to single sign-on. The user authenticates to the passport infrastructure, either directly through www.passport.com or through an affiliate site that makes use of functionality provided by passport.com. Once the user is authenticated and authorized by Passport, their authentication status is also available to other Web Services that use Passport. Like SAML, this provides single sign-on. However, the model is different, relying on a central point of authentication rather than SAML's architecture where authentication happens at an individual Web Service. By being implemented at the site of the Web Service itself, SAML authentication and authorization information may be based on *role-based security*. Role-based security means that access to resources is based on the user's organizational role; for example, in a medical setting doctors may have access to certain information while nurses have access to different information.

Another industry proposal for the SSO on the Web is the Liberty Alliance Project, championed by Sun. The Liberty Alliance Project aims to enable a noncentralized

approach to SSO, termed a “federated network identity.” At the time of this writing, it appears the Passport proposal by Microsoft may be taking a similar tack to the Liberty Alliance Project.

PKI for Web Services: Introducing XKMS

As you may recall from Chapter 2, PKI is a system that allows public key keys to be trusted by providing key signing and key validation services. Although accepted as an important, even vital, technology, PKI has a reputation for being notoriously difficult to implement. The benefits of XML and Web Services apply quite naturally to PKI: addressing interoperability and integration issues. The XML Key Management specification (XKMS) enables PKI services such as trustworthily registering, locating, and validating keys through XML-encoded messages. Because XKMS is service-oriented and uses XML messages, it is only natural that it be implemented as a SOAP-based Web Service giving it the distinction of not only being useful for securing Web Services, but also being available as a Web Service itself. By leveraging the benefits of XML and by learning from past experiences with pre-XML PKI architectures, XKMS makes PKI practical for common use.

Like XML Signature, XKMS eliminates the need for ASN.1 functionality in software that deals with digital certificates. It goes further, however, and can allow XML software to use digital certificates and PKI without the need to implement cryptography algorithms. This is useful for software developers, many of whom may not have the time or inclination to delve into cryptography or employ cryptography toolkits.

WEB SERVICES SECURITY THREATS

We’ve seen the positive side of Web Services security: the industry cooperation on new specifications and frameworks. Now let’s investigate the negative side. The new specifications that implement the principles of security for Web Services are useless if the user is required to “play by the rules.” A sophisticated authorization system using SAML and WS-Security is useless if the Web Service on which it runs can be disabled by a CodeRed attack. (CodeRed is a worm program that attacks IIS Web servers.)

Some of these “new threats” are, in fact, old threats, such as buffer overflow and attempts to exploit other programming errors, but the avenue of attack—SOAP—is new.

Firewalls have traditionally addressed vulnerabilities at the lower layers of the OSI stack. Firewall functionality has progressively moved up the OSI stack to reach the application layer. However, they are not yet “SOAP-intelligent.”

The following section examines these aspects of Web Services security.

Web Application Security

Application layer security existed long before SOAP. Application layer security for Web servers involves securing both the Web server itself and Web applications that use

the Web server as their platform. A Web application is a CGI-based application with which the user interacts using a Web browser. Attacks on Web applications initially focused on attacking the platform itself, exploiting security holes in the Web server. These frequently took the form of buffer overflow attacks. Like the “ping of death” attack we saw at the network layer in Chapter 2, buffer overflow attacks on a Web server presented more data than the Web server expected. This data would be written to memory, and could find its way into the execution stream. This allows arbitrary commands to be executed on the server.

It is difficult and time-consuming to produce a buffer overflow attack, but once produced, the attack can be packaged into a scriptable tool that so-called “script kiddies” can use. Script kiddies use existing techniques and programs or scripts to search for and exploit weaknesses in computers on the Internet. The derogatory nature of the term refers to the fact that the use of such scripts or widely known techniques does not require any deep knowledge of computer security.

Gradually, buffer overflow attacks on HTTP implementations were addressed in patches to Web servers. At that stage, attacks began to exploit the extra features bundled with certain Web servers, features often installed whether users wanted them or not. These extra features included indexing engines and example scripts. After these holes were patched, it became more difficult for hackers to construct attacks on Web server software. This is when application layer hacking attacks progressed to attacking Web applications, rather than the platforms on which the Web applications run. These are not across-the-board attacks that can be packaged and used against thousands of Web servers by script kiddies. These attacks are specific to individual Web applications. However, they can be put into categories, including the following:

- **SQL attacks** Inserting SQL statements into Web forms in order to force a database to return inappropriate data, or to produce an error that reveals database access information. For Web Services, this category of attack translates to manipulating data in a SOAP message to include SQL statements that will be interpreted by a back-end database.
- **Directory traversal attacks** Attempts to bypass hyperlinks by attempting to directly access resources. For example:
 - If a URL is `http://www.example.com/documents/sales.htm`, what happens if `http://www.example.com/documents/` is requested?
 - Does a directory called `/test/` exist?

For Web Services, this category of attack translates to attempting to detect other SOAP services which are not explicitly offered.

- **URL string attacks** Manipulating CGI name/value pairs in the URL string; for example, changing “`maxResults=10`” to “`maxResults=1000`” to return more information from a database. For Web Services, this translates to circumventing the rules on SOAP parameters (for example, if a search SOAP service takes an integer between 1 and 10 as a SOAP parameter, what if the number 1000 is submitted?).

Many of these attacks can be avoided by implementing careful programming practices and by cleaning up resources that are not required on the Web server. However, it is often the case that no matter how much care is taken; vulnerabilities can slip through the net.

TIP In order to guard against the possibility of attacks on application vulnerabilities, consider the use of an XML firewall or XML proxy to filter SOAP requests before they reach your application.

When bound to HTTP, SOAP itself can be seen as a Web application, albeit a more standardized and formalized application than what has gone before. It is likely that initial SOAP implementations will be vulnerable to attacks based on invalid data such as buffer overflow attacks or attacks based on SOAP routing that attempt to create SOAP “worms.” This vulnerability is not necessarily due to carelessness but, rather, due to the fact that all security bases cannot be covered in initial versions of any software. The lesson from the history of Web server attacks is that once the platform is secured, the playing field shifts to the applications implemented on that platform. These attacks are potentially more dangerous than in the case of CGI applications; because of the nature of the business processes implemented using SOAP.

The Role of Firewalls for Web Services

“SOAP bypasses firewalls.” This phrase is frequently heard. Let’s examine what it means. The first question to ask is: What is a firewall? The answer is that different categories of firewalls apply to different layers of the OSI stack.

Packet-Filtering Firewalls

The lowest layer at which a firewall works is layer 3. At this level, the firewall checks if the information packets are from a trusted source and is not concerned with the content of the packets. These are called “packet-filtering firewalls” and are usually part of a router. IP packets are compared to a set of criteria and dropped or forwarded accordingly. These criteria can be source and destination IP address, source and destination port number, the protocol used, and the format of the IP packet.

The “ping of death” that we encountered in the previous chapter applies at the network layer and is protected by packet-filtering firewalls. Much of this functionality is now built into operating systems and routers.

Circuit-Level Firewalls

At layer 4, firewalls filter traffic based on more sophisticated criteria. TCP layer firewalls are known as “circuit-level firewalls.” These firewalls monitor TCP handshaking to determine a session’s legitimacy. Information about the protected network they are protecting is hidden, because packets appear to originate from the firewall and not from an address inside the protected network. These firewalls do not filter individual packets; rather, filtering is based on the rules of the TCP session, including who initiated the session and at what time.

Circuit-level firewalls prevent “session hijacking”—sending an IP packet that is intended to appear as if it belongs to a trusted session. It also hides an internal network from an attacker who wishes to scan it for vulnerabilities.

Application-Level Gateways

Application-level gateways filter packets at the application layer. They are aware of what traffic meant for specific applications should look like. For example, it knows the difference between Web traffic and telnet traffic, even though both use TCP/IP. Application-specific commands and user activity can be logged. These firewalls are relatively processor-intensive.

An application-level gateway will know that if it is protecting an e-mail POP server, the command “USER” is allowed, and takes one parameter (that is, the username). Anything else is not allowed. For example, traffic that looks like telnet traffic directed to the POP server will be blocked.

Stateful-Inspection Firewalls

Stateful-inspection firewalls operate at multiple levels and include much of the functionality of packet-filtering firewalls, circuit-level firewalls, and application-level gateways. These are complex and powerful, but tend to be difficult to configure. When not properly configured, they may contain security holes.

Application Layer Firewalls

Many firewalls have been configured to only allow Web (HTTP, SSL) and e-mail (POP, SMTP) traffic to pass. Other TCP/IP ports, and other protocols, are routinely blocked. It has become standard practice to “tunnel” other applications through the Web ports (80 for HTTP, 443 for SSL), effectively disguised as normal Web traffic. This is generally not done for malicious reasons, but rather for pragmatic reasons, because all other ports are blocked. In particular, SOAP is very frequently bound to HTTP. It provides a standardized means for applications to communicate over the Web ports. One of the lessons of computer industry history is that standards drive usage and SOAP promises to enable the explosive growth of application communication over the Web ports.

When a firewall examines a SOAP request received over HTTP, it might conclude that this is valid HTTP traffic and let it pass. Firewalls tend to be all or nothing when it comes to SOAP. A SOAP-level firewall should be able to:

- Identify whether the incoming SOAP request is targeted at a Web Service that is intended to be available.
- Identify whether the content of the SOAP message is valid. This is analogous to what happens at the network layer, where IP packet contents are examined. However, at the application layer it requires knowledge of what data the Web Service expects.

Content-Filtering Security at the Application Layer

Web Services present a new avenue of attack into the enterprise. Even so, some of the tactics are familiar: feeding unexpected data to an application in order to confuse it, or

disable it. In Chapter 2 we saw how the “ping of death” was an attack that operated at the network level to provide unexpected data in an Internet Protocol (IP) packet. Web Services present details of their interface in WSDL files, which effectively say, “Here are the details of the data that I expect.” This invites a hacker to send it inappropriate data in order to see what happens.

A WSDL file may contain the following line:

```
<xsd:element name="tickerSymbol" type="string" />
```

This indicates that one of the parameters expected by the Web Service is a string, called “tickerSymbol.” The options for a speculative attack on this Web Service would include sending it a number instead of a string, or sending it a very large string designed to overload the Web Service. It is important, therefore, that “sanity checks” are performed on incoming data directed to Web Services. This may take the form of checking SOAP parameters against an XML Schema. However, XML Schema validation is processor-intensive. In addition, certain portions of a SOAP message may be *volatile*, meaning that they change while in transit between the SOAP requester and the Web Service. Volatile portions of a SOAP message include the header, which may contain routing information that changes as the message is routed. Therefore, it is more appropriate to use XPath to narrow down the data validation to nonvolatile portions of the SOAP message.

Another aspect of content filtering is ensuring that only valid Web Services are called. Firewalls must be able to distinguish SOAP requests from invalid requests. A valid request and an invalid request may differ only on the basis of the SOAP method called. The following code listing shows a valid SOAP request to a method called “GetTime” that takes a time zone as a parameter:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<SOAPAPP:GetTime xmlns:SOAPAPP="http://tempuri.org/message/">
<TimeZone>GMT</TimeZone>
</SOAPAPP:GetTime>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following code listing shows a SOAP request that targets another Web Service method, called “ResetComputer”:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<SOAPAPP:ResetComputer xmlns:SOAPAPP="http://tempuri.org/message/">
```

```
</SOAPAPP:ResetComputer>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

The challenge for firewalls is how to allow the first SOAP message, targeting a valid method of a Web Service, and to block the second one. There are analogies with firewall functionality at lower layers of the OSI stack. Filtering on the targeted SOAP method is only the first step, however.

Filtering on the data that is provided to a Web Service is more complicated. The details of each Web Service are specific. Consider a Web Service that takes a ZIP code as a parameter. The valid input is a five-character string. If the Web Service receives 5,000 characters as input, this may indicate that an attacker is testing for vulnerability to a buffer overflow attack. In order to block this sort of attack, a firewall must be aware of what type of data is valid for the Web Service.

The Next Steps for Firewalls

Application layer security began by ensuring that connecting entities played by the rules for applications that ran over TCP/IP; then, they progressed to patching holes in Web servers; and now it has moved to blocking attacks on Web applications. A SOAP implementation bound to HTTP may be seen as a Web application itself, and therefore the next step is to prevent SOAP implementations from attack. After protecting the SOAP implementation, the final step is to protect Web Services that use that SOAP implementation.

The challenge is to ensure that the firewall rules are in sync with the Web Services themselves—and it seems obvious that UDDI and WSDL should be used for this purpose. UDDI, after all, is used to return a list of deployed services. A dynamic updating firewall could query this list using a UDDI query such as we encountered in Chapter 1, and use this to ensure that only legal traffic passes through.

The next challenge is to ensure that only permitted traffic travels *out* of the network to third-party Web Services. A number of organizations are investigating the establishment of two-way Web Services gateways that act as “choke points” for SOAP traffic. This ensures that only valid SOAP traffic comes into the enterprise, and only valid SOAP traffic leaves the enterprise.

This is the natural evolution of firewall functionality, which began at the network layer and has been “climbing the ladder” of the OSI stack ever since.

Hint: When choosing a firewall product, check if the vendor supports SOAP filtering. A firewall should be capable of blocking SOAP messages based on target (endpoint) and based on the payload of the SOAP message, validated against an XML Schema.

This is the end of Part I. We have seen what Web Services are, what the high-level principles of security are, and how security is applied to Web Services. Part II looks at the implementation technologies in-depth.